

Problems in Modern High Performance Parallel I/O Systems

Robert L Cloud¹

The University of Alabama at Birmingham
Department of Mechanical Engineering

September 6, 2011

¹rcloud@uab.edu

0.1 Abstract

In the past couple of decades, the computational abilities of supercomputers have increased tremendously. Leadership scale supercomputers now are capable of petaflops. Likewise, the problem size targeted by applications running on such computers has also scaled. These large applications have I/O throughput requirements on the order of tens of gigabytes per second. For a variety of reasons, the I/O subsystems of such computers have not kept pace with the computational increases, and the time required for I/O in an application has become one of the dominant bottlenecks. Also troublesome is the fact that scientific applications do not attain near the peak theoretical bandwidth of the I/O subsystems. In addressing the two prior issues, one must also question the nature of the data itself; one can ask whether contemporary practices of data dumping and analysis are optimal and whether they will continue to be applicable as computers continue to scale. These three topics, the I/O subsystem, the nature of scientific data output, and future possible optimizations are discussed in this report.

Chapter 1

Introduction

1.1 Static I/O Performance and Rapidly Scaling Computation

A 2005 paper from Lawrence Livermore National Labs [6] states that ASCALibre supercomputers, i.e. leadership class machines capable of running the most demanding applications, require I/O bandwidth of 1 GB/sec per teraflop of computing capability. This rule would enable a 100 teraflop supercomputer to perform sustained writes of 100 GB/sec, a benchmark demonstrated that year by ASCI Purple.

Since 2005, the computational abilities of the most powerful supercomputer have increased by two orders of magnitude, from 100 teraflops to nearly 10 petaflops (taking K, June 2011s #1 as the example). At the same time, the total I/O throughput for these systems, measured in the total theoretical bandwidth for writes (the sum of the throughputs to each object storage target (OST) in the parallel file system) has remained nearly static. As this is being written, K writes across 864 OSTs to give realized I/O throughput of 96 GB/s [12] using the benchmarking software IOR [6].

This I/O plateau is not only observed with K; a 2008 paper [3] gives the results of very detailed testing of a Lustre based file system on a leadership scale computer, Jaguar of ORNL, a Cray XT. The system could perform computations in excess of 250 teraflops at the time, and the theoretical peak performance of the I/O system was 72 GB/s. The authors used MPI-IO based benchmarks to measure the actual performance of the filesystem and found that less than half of the theoretical peak could be achieved by such

an approach. The poor actualized performance of parallel file systems with common I/O libraries is explored later in this report.

This report aims to make apparent the impending I/O challenges that will be faced by high performance computing systems. Modern scientific software development programs aim to create tools that will be in use for decades. Necessarily, we must take into consideration the hardware upon which such tools will be executed. By introducing ideas taken from recent research in I/O into our software, we can ensure that it scales into the petaflop level and beyond.

Furthermore, it is inevitable that our software will be run on a diverse selection of hardware with varying filesystems and performance characteristics. It is thus worthwhile to explore ways through which we can minimize the necessary effort to port the software across platforms. Recent research has been directed at this goal and should be brought to attention.

1.2 I/O in the High Performance Environment

At scale, file system I/O is primarily used for application checkpointing. Writing to the filesystem dominates reading from it by a factor of 5 [6]. As noted in [9], the increases in computational performance of HPC systems can be to a large degree attributed to the use of more hardware components. As the complexity of the system increases, the average time to failure falls. Large scale modern machines, such as a 100,000 node BlueGene/L at LLNL have a failure once every 8 hours. In the future, at exascale, machines may have failure rates on the order of minutes. Furthermore, as the number of nodes increases, the amount of data that needs to be written with every checkpoint grows exponentially. Thus we have a need for increasing checkpoint frequency as well as more data to be written at each checkpoint.

1.3 Data Intensive Applications and Scaling I/O

Data intensive applications have been enabled by the availability of powerful parallel file systems and parallel I/O libraries which enable applications

to write out many terabytes of data at rates measured at up to 100 GB/s. However, as noted in [8], data intensive applications are straining the capabilities of even the most powerful of file systems. This problem will be even greater in the future as the number of compute cores in large scale systems will continue to rise at a rate much faster than the number of striped file targets in the parallel file system, primarily because of the expense in I/O hardware. Furthermore, parallel file systems rely on centralized metadata storage which can become overwhelmed by the network traffic required to communicate with many thousands of I/O clients.

Optimizations to the parallel I/O system are considered difficult. These data intensive scientific codes access the I/O subsystem in ways that are not particularly amenable to parallelization. Frequently, these accesses take the pattern of many small, noncontiguous accesses of a single large file. As noted in [10], this produces effective serialization of accesses as file locking semantics are required to maintain consistency and the guarantee of correctness. Parallel I/O frequently consists of many small atomic accesses.

[3], after a study on the I/O needs of developers of extreme scale applications, found that the programmer must increasingly become familiar with the intricacies of the I/O subsystem in order for their application to continue to scale. It is no longer sufficient to expect the parallel I/O libraries to continue working as they have in the past and provide sufficient performance with merely the addition of new hardware on the storage side. The scaling of I/O is currently at an impasse. One cannot continue with the ways of the past and expect the system to work as the number of cores in a computer system scales into the millions. One temporary, and necessary, stopgap in scaling I/O is data staging, further discussed below. This approach, now common, forwards the data from the compute nodes to a number of processes designated for I/O. In this way, network traffic between the clients and the file server is reduced and greater performance can be demonstrated.

1.4 I/O as it Relates to Modern Software Development Programs

Modern scientific software development programs that seek to develop codes that will remain in production for decades must address the scalability problem. For our purposes, one can concentrate optimizing I/O in homogeneous,

secure computing environments, which are the most common in leadership scale computing today. The leadership scale computers of today will within years become commonplace and the I/O challenges currently faced by the developers of the largest computers will need to be addressed by all.

Chapter 2

Contemporary Approaches to Optimizing HPC I/O

2.1 Data Staging

2.1.1 Introduction

The problem of I/O in the high performance environment is well recognized. Previous work has been done to address it on a variety of fronts. Important to the discussion is [3] which describes detailed tests run to analyze the performance of a Lustre parallel file system on Jaguar of ORNL. It introduces the concept of subsetting, derived from the observation that higher performance can be had by only using a fraction of the total number of processes to interact with the file servers. This can be attributed to less contention in communicating with the OSTs and that, generally, fewer, larger interactions with the file system are better than many smaller ones. An additional contribution of this paper is that it compares the performance of MPI-IO to HDF-5, an I/O framework that provides a higher level of abstraction, and the paper finds that write performance between the two can be comparable.

These findings can be generalized into the notion of data staging. Rather than having all client nodes directly interacting with the file system servers, they forward their data and I/O requests to intermediary processes. Major work has been done in this area by the Center for Ultra-scale Computing and Information Security at Northwestern University and Georgia Techs Center for Experimental Research in Computer Systems (CERCS).

2.1.2 Delegate Caching

[10] has developed a way to use what they term I/O delegates as intermediaries between the processes and the compute nodes. They observe that the BlueGene architecture has built in a layer between the compute nodes and the I/O servers through which requests can be funneled and draw inspiration from this in creating their prototype. Their system treats independent I/O the same as collective I/O, and they find that by using it, independent I/O can reach double the write performance of native collective I/O, an impressive achievement as collective operations have been traditionally better performing than independent I/O.

Her work addresses the problem of serialization, common in independent I/O operations. As several processes contend to write to the same file, the file system must lock it and contending processes must be queued. By reducing the number of clients, there are fewer file locks. Furthermore, they implement a caching mechanism that reduces the number of I/O operations required by coupling smaller requests into a few larger ones.

2.1.3 Flexible I/O Forwarding

[2] again reiterates the decline in I/O throughput compared to computational ability. They note that an increasing number of applications are becoming I/O bound and in the future this number will inevitably grow. Furthermore they state that the main difficulty in contemporary I/O research is to achieve maximum throughputs for existing architectures while also scaling up the application. One major problem in doing this is that the major modern parallel file systems, e.g. Lustre, PVFS, GPFS and PanFS were developed for smaller systems than what they are currently being used for. They also discuss the various levels at which enhancements to the I/O subsystem could be made, e.g., at the level of the parallel file system, at the level of the I/O library (generally ROMIO), or at the level between the two, the I/O forwarding layer, for which they also draw inspiration from the BlueGene systems. The forwarding they present would be transparent to the application and I/O libraries and work on multiple architectures, file systems and interconnects.

2.1.4 DataTaps and I/O Graphs

[13] looks at the use of DataTaps to add some structure to data which enable the data to be decoupled from the application in space and time. This header information is extremely lightweight, thus DataTaps can also be used as a substitute for typed I/O libraries such as HDF5 and NetCDF without their characteristic performance penalty. A smaller number of DataTap servers get the structured data off the compute nodes. From the DataTap servers, IO graphs are used to route the data to disk.

2.1.5 Irregular Accesses to Noncontiguous File Sections

Scientific applications are not able reach I/O throughputs near the theoretical peak bandwidth of the I/O subsystems. This is because scientific applications do not typically make I/O requests in the optimal fashion. While some have been able to utilize collective I/O operations, as [10] points out, more and more are behaving erratically and requiring independent I/O which has been traditionally provided less throughput than collective I/O.

[8] directs its attention to the problem that scientific codes do not make best use of available I/O capacity. He says this is because scientific codes make I/O requests to a large number of non contiguous regions, with each request being of high latency. He states that most research in parallel I/O has been in attempting to optimize such requests. He argues that one can sidestep the issue by discarding the view of a file as a linear series of bytes and rather looking at data to be output as unrelated objects.

Lustre is the most common parallel file system for the largest supercomputer systems today, and it services 15 of the worlds top 30 computers(as of July 2011). As evidenced in [3], MPI-IO is poorly supported by Lustre, and [8] discusses the reasons for this in a large section of the paper. The primary problems, he claims, are that Lustre primarily works within the POSIX I/O view and thus poorly supports possible parallel optimizations. He demonstrates that the assumption that large, non-contiguous I/O requests are not necessarily the best way to attain performance but can rather lead to very poor throughputs. Also problematic for parallelization is that Lustre strictly enforces file consistency semantics, i.e. it locks files to protect from concurrent writes to the same region. These semantics limit performance and can be worked around by instead viewing the data to be output as unrelated

objects.

2.1.6 Interval I/O

[8] is a dissertation that develops what they term Interval I/O, another method to add an intermediary layer between the computation and I/O servers. Primarily designed for multiple processes writing the same file, Interval I/O intelligently partitions a file into intervals using application access patterns. The benefits are similar to that described in [10] in that it reduces contention and serialization, especially for atomic operations.

One important contribution of this work is that it argues that the view of a file as a flat contiguous series of bits is one of the most important contributors to poor parallel I/O performance. Rather, one should consider the data to be written out as discrete objects. If one were to do this, less contention would be had and greater parallelization made available.

2.1.7 Conclusion

The works above have in common the agreement that some intermediary layer must be added between the computational nodes and the I/O servers for continuing scalability. They agree that contemporary parallel file systems will be a bottleneck as the number of compute cores continues to scale. These data staging techniques presented are similar in that they all use a subset of processes to communicate with the file servers in order to reduce contention and communication. They differ in their respective implementations of that idea, specifically the level at which the data staging functionality is added. The level of implementation can be kernel space or user space, the implementation can either provide an application access to an API or it can be transparent to the application and forward the calls automatically. [8], a dissertation dated December 2010, provides a discussion of the differences in the various approaches to performing data staging.

Data staging, however, is only a partial, and temporary, solution to the problem. The number of I/O delegated processes must increase along with the number of computational cores. As long as a central metadata server is required, there will be write contention. It would require a radical departure from the reigning model address this fundamental problem. Such an idea is reported below in chapter 3

2.2 Decoupling I/O From The Application Core

2.2.1 Introduction

An important fact about production scientific codes is that they have a long lifespan. Once they have been thoroughly debugged, any modifications to the code must be accompanied by another review to ensure correctness. While the code may fundamentally stay the same over decades, the systems upon which it runs may change dramatically. Different institutions may also run the same code on different hardware organizations.

One problem in the portability of code is that performance may vary dramatically between different parallel file systems. It is understood that, e.g., Lustre performs especially poorly with the MPI-IO library, and [8] addresses a significant portion of his dissertation in the attempt to understand why. Other file systems handle I/O libraries with varying amounts of competency. By changing the I/O fundamentals of an application, it is possible to port it to computers with other I/O subsystems, but doing this within the source is burdensome and costly in man hours and requires maintaining multiple branches of a code.

One can decouple I/O from the applications core compute functionality by using abstraction. One makes calls to a middleware library which then maps the I/O to the particularity of whatever file systems the application happens to be running on. This decoupling is primarily for human convenience. This abstraction is present in the ADIOS I/O middleware.

[13] discusses two more notions of decoupling I/O. One can decouple I/O applications temporally and spatially. Spatial decoupling is essentially data staging, i.e., having I/O requests forwarded from the compute nodes to nodes delegated for I/O. Temporal decoupling is similar to caching, combining multiple small I/O requests into fewer larger ones. Both notions are also present in [10] which uses delegates to decouple I/O spatially and a caching mechanism for decoupling I/O temporally.

2.2.2 ADIOS

Researchers at Georgia Tech have developed a middleware system to enable applications to change the underlying I/O operations without making changes to the application source [7]. One can program the application with high level I/O calls derived from POSIX and then use a configurable XML

file to map those calls to what is optimal for the underlying hardware. This API, named ADIOS(Adaptable IO Systems) has been tested on leadership scale machines such as Jaguar and has achieved I/O throughputs comparable to that of lower level direct MPI-IO based benchmarks.

2.2.3 Exploiting Latent I/O Asynchrony

[13] also addresses the need to decouple I/O from the application, and they look at the tolerance that an application has in decoupling I/O both temporally and spatially from the core computation. They note that the tolerance for asynchrony is a property of HPC codes that hasnt been fully explored, but one can achieve significant performance improvements by overlapping computation with I/O.

They use DataTaps to add headers, structuring the data, enabling it to be decoupled in space and time. DataTap servers pull the data from the DataTap clients. This is done during periods of relatively low activity within the interconnect to avoid interfering with application performance on the computational side.

IOgraphs buffer the data, send it to special I/O nodes, and eventually write it to disk in structured form. Metabots further decouple I/O temporally by transforming the structured data on disk into a more regular form. This operation can be done at anytime spare CPU cycles are available.

This system integrates with the ADIOS API to achieve another level of decoupling, that of indirection. It has been tested on large scale systems at Oak Ridge with applications such as GTC, a particle based fusion simulation that scales to hundreds of thousands of cores and must output tens of terabytes of data with each checkpoint. They describe that GTC can have bursts of output that exceed the bandwidth of the I/O subsystem, but with proper use of IOgraph buffering this can be overcome, allowing GTC to begin the next computation while the data is being written to disk. Their results also indicate that DataTaps can structure data at near peak bandwidth.

2.2.4 Overlapping I/O with Iterative Solvers

[4] also explores decoupling the I/O from the application in the context of an iterative solver. Rather than waiting on the I/O for each iteration to complete before beginning the next, they decouple the I/O temporally and move it to I/O nodes while continuing on iterating the solver. The result is

perceived I/O bandwidth of the sum of the memory speed to the I/O nodes, or 21 TB/s in the example application they give.

2.2.5 Conclusion

It is strongly recommended that developers of applications that may be used on varying hardware consider adapting such an approach to decrease the future amount of time required in maintaining the source code. By using such a mechanism, one can better future proof ones code and allow the application to be adapted to yet to be developed file systems. Furthermore, by overlapping I/O with core computational functionality, one can achieve much higher bandwidth, both realized and perceived by the application.

Chapter 3

Promising I/O Developments for the Future

3.1 Introduction

[1], a dissertation dated 2009, states that the problem in I/O scalability is worsened by contemporary parallel file systems and that the central server paradigm cannot handle the I/O requirements of modern data intensive applications. Furthermore, scientific applications increasingly depend on data stored in remote locations. Given examples of applications dependent upon dispersed data storage include the Large Hadron Collider, which generates data on the order of petabytes and because of the collaborative nature of the project, this data must be shared across wide area networks.

These developments, he argues, call for a reevaluation of the current approach to I/O, one that ensures the scalability of applications to millions of cores and beyond and takes into account the often distributed environment in which the applications requiring the most compute capability will operate.

[11], a very recent paper dated June 2011, states that the network storage based paradigm will no longer service exascale computers with billions of computational threads and that, if it were attempted, there would be a performance collapse. The mean time to failure would fall below the time required to checkpoint, and concurrent communication from clients would overwhelm the metadata server. Rather, they propose that advances in storage technology will enable distributed metadata services on the compute nodes.

They point out that the gap between computational capacity and I/O bandwidth has seen a 10x increase over the past decade. They argue that the most significant barrier to building exascale computers will be the I/O subsystem. Common I/O operations, when performed by millions of concurrent processes, will take more time than the mean time to failure. Even the action of booting the machine is projected to take 7 hours.

They argue that distributed file systems should exist in tangent with classical parallel file systems and be used for the jobs requiring the full computational capacity of the machine. Every node would be equipped with solid state storage and use the multicore capabilities to participate in the management of both metadata and data. They note that this approach was previously unfeasible because of the high rate of failure in mechanical magnetic storage devices, but new developments in solid state storage may attain reliability sufficient. The authors are working on a prototype of such a distributed storage system named FusionFS.

3.2 Questioning the Checkpointing Paradigm

As noted earlier, checkpointing dominates the I/O for large applications, and writes can dominate reads by a factor of 5. As the number of compute nodes scale, the total amount of system memory increases and thus the amount of data that needs to be dumped at each checkpoint is greater. Furthermore, as the hardware complexity of computer systems increase, the mean time to failure falls, and the frequency of checkpointing must rise.

Some researchers have sought to address the scalable I/O problem by questioning this checkpointing paradigm. Currently the practice is to have each compute node dump the same large subset of the data it holds in memory to disk at every checkpoint. The data written must be sufficient to restart the application in the event of a failure.

[9] introduces a multi level checkpointing system. They analyze the frequency of different types of failure in an HPC system and observe that the most common types of failures do not require a full heavyweight checkpoint. Rather, the most common failures only disable a node or two at one time and that failures of multiple nodes happen predictably. They report that 85% of failures disable at most one compute node in the cluster. By writing checkpoint data to RAM, flash memory, or scratch disk space on the compute nodes, these failures can be protected for while achieving effective

checkpointing speeds greater than two orders of magnitude faster than the existing model. The fewer, less frequent heavyweight checkpoints protect against multinode failures.

One drawback of this approach is that it requires storage on the compute nodes. This would either mean one would use a portion of the ram or have some kind of non-volatile storage available on each node. [11] argues that systems at the exascale will indeed have such components and will have to make use of them for file system data management. Some combination of multi level checkpointing with distributed parallel file systems at the node level may be a viable solution for future scalability.

Diskless checkpointing is the use of volatile memory to add levels of redundancy by storing just enough information to allow for the recomputation of the original data in case of a node failure. Analogous to RAID technology that is instrumental in affording redundancy of disk drives, diskless checkpointing stores parity bytes in the memory of other nodes that, when added to the data held by other nodes, allows for data recovery when one node goes down. While this would eliminate the need to store checkpointing data to the parallel file system, it has traditionally been too computationally expensive to consider. However, novel architectures are becoming more prevalent in HPC environments and heterogeneous compute nodes with highly parallel accelerator cards are now common. One can leverage the highly parallel nature of e.g. GPUs to accelerate Reed-Solomon computation of the parity information and make diskless checkpointing practical.

[5], to be published in the November 2011 Supercomputing proceedings, develops a very high frequency low overhead checkpointing system using heterogeneous components. Noting also that post petascale systems will have very small mean times to failures and extremely large amounts of data to write out(hundreds of teraflops to petabytes), they argue that it is essential to develop a novel approach to checkpointing. They discuss the most modern supercomputing systems, noting that they have solid state memory or some other type of non volatile storage space on the compute node.

While soft errors can always be recovered by the data stored on the compute node, even hard errors, i.e., the full node going down, can be recovered from if they make use of partner nodes to replicate erasure codes(typical RAID-like parity byte encoding). This requires an extra transfer of the code but is more efficient than using the central parallel file system to protect against hard failures. This approach does require that one takes into consideration the topology(spatial arrangement) of the nodes, as one doesnt want

to store parity bytes on partner nodes that fail predictably together.

They build upon the multi level checkpointing system discussed above, but theirs contains three levels. The first level protects against soft errors by storing checkpoint data locally on the node. The second level uses heterogeneous Reed-Solomon coding to create parity bytes and stores them on partner nodes. This can be used to recover from the most common type of hard failures. The third level and least frequent writes checkpoint data out to the parallel file system. This protects against catastrophic system failures. The primary focus of their paper is the second level, or the use of heterogeneous architecture to protect against limited hard failure.

One more interesting point they cover is that as the fabrication process decreases in scale, e.g., from 90 nm to 16 nm, the rate of soft errors increases significantly. This observation leads to a discussion of pattern prediction in failures which enable intelligently developed redundancy systems.

3.3 Are Parallel File Systems Really Parallel?

[2] notes that modern parallel file systems were really developed for smaller supercomputing systems than what they are currently being used for. [10] states that achieving scalability is simply not possible without a fundamental change to the premises upon which they operate. What one considers as parallel distributed file systems are in fact not really distributed, but rather depend on a central metadata server to drive the requests. Communication with this server can become saturated with too many client processes and performance can fall dramatically.

The performance improvements that data staging brings are because what we commonly understand to be distributed file systems are fundamentally an outgrowth of traditional serial file systems. However, very recent developments in new hardware, some researchers are questioning whether it would be appropriate to implement a fully distributed file system, one that is no longer dependent upon a central metadata server.

Both [1] and [11] develop models of distributed parallel file systems that are not dependent upon a central metadata server. [1] is applicable to large scale collaborative scientific computing that relies on resources distributed across wide area networks. [11] advocates a distributed file system using

contemporary developments in solid state storage hardware which offers resiliency levels previously unmet by magnetic disk drives. With such resiliency, one can have each compute node participating in metadata and data management rather than having it managed by a redundant central metadata server.

3.4 In Situ Analytics

Petascale applications generate data of such tremendous scale (on the order of tens of terabytes per checkpoint or iteration) that the post processing of it becomes extremely time consuming. Data output has reached such a level that, according to Dr. Karsten Schwan of Georgia Tech, if one does not perform some kind of analysis on it while in transit, it is unlikely that one can fully process it. In an application one always makes the choice of what subset of data to output. With in situ analytics, one can intelligently discover interesting parts of data on the fly while the application is running. Using such techniques enable us to stop or correct an erroneous run, manipulate or refine a mesh, or provide input to guide an application to a more stable solution.

The in situ analytics of petascale applications are being studied by a number of labs. One of the most prominent is the visual study of combustion at Sandia [14]. This is case study of using in situ visualization using the direct numerical simulation of turbulence at small scale. While scientists have traditionally tried to dump as much raw data as they possibly can for post processing analysis, in situ analytics attempts to tame the flood of data as it is in transit and prepare it for visualization while the application is being run.

At the petascale, it is very difficult to post process data because one does not typically have petabytes of data storage available and the transfer of such amounts of data would be prohibitively expensive. While one can only transfer and examine certain time steps, this defeats the point of high resolution simulations in the first place. [14] argues that the most effective way to process the great amount of data is directly on each compute node in situ. By doing this we save both the cost of transfer time of data and the time required to post process it. They address the unique challenges of doing visualization in situ, and they find that visualization calculations only take an acceptably small fraction of the total supercomputing resources.

Chapter 4

Conclusion

4.1 Levels of Indirection and Planning for the Future

All problems in computer science can be solved by another level of indirection states the famous aphorism in computing. For new development projects, we must ensure that our applications scale into the petaflop level and are easily portable to other hardware architectures by considering the research presented in this report.

The ADIOS API, currently being maintained by Georgia Tech and Sandia National Labs provides a simple, POSIX based interface for cross platform I/O programming. By modifying and tweaking an accompanying XML file, we can get high performance using a variety of I/O libraries which would make our applications more portable to other file system architectures.

By integrating such an approach with advances in data staging, we can make sure that our application can scale into the tens or hundreds of thousands of cores without overwhelming the central servers in a parallel file system. By exploiting latent asynchrony, or the tolerance for an application to overlap computation and ancillary I/O operations, we can achieve much higher perceived throughputs to the I/O subsystem.

Lastly, as we are developing applications that should remain in use for the next few decades, we should at least consider the new research in both multi level checkpointing and the potential for disruptive new revolutions in the fundamental architecture of I/O subsystems.

Bibliography

- [1] Nawab Ali. *Rethinking I/O in High-Performance Computing Environments*. PhD thesis, The Ohio State University, 2009.
- [2] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and P. Sadayappan. Scalable I/O forwarding framework for high-performance computing systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.
- [3] M. Fahey, J. Larkin, and J. Adams. I/O performance on a massively parallel Cray XT3/XT4. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, 2008.
- [4] J. Fu, N. Liu, O. Sahni, K. E. Jansen, M. S. Shephard, and C. D. Carothers. Scalable parallel I/O alternatives for massively parallel partitioned solver systems. *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.
- [5] L. B. Gomez, Dimitri Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka. FTI: high performance Fault Tolerance Interface for hybrid systems. In *Supercomputing 2011*.
- [6] Richard Hedges, Bill Loewe, T. McLarty, and Chris Morrone. Parallel File System Testing for the Lunatic Fringe: the care and feeding of restless I/O Power Users. In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, number Msst. Published by the IEEE Computer Society, 2005.
- [7] J.F. Lofstead, S. Klasky, K. Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io

- system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24. ACM, 2008.
- [8] Jeremy Logan. *Improving Parallel I/O Performance Using Interval I/O*. PhD thesis, The University of Maine, 2010.
 - [9] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design , Modeling , and Evaluation of a Scalable Multi-level Checkpointing System. In *Supercomputing 2010*, number November, 2010.
 - [10] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Delegation-based I/O Mechanism for High Performance Computing Systems. In *IPDPS 2011*, 2011.
 - [11] Ioan Raicu, I.T. Foster, and Pete Beckman. Making a case for distributed file systems at Exascale. In *LSAP 2011*, pages 11–18. ACM, 2011.
 - [12] Shinji Sumimoto. An Overview of Fujitsu’s Lustre Based File System. Technical report, Fujitsu, 2011.
 - [13] P. Widener, M. Wolf, H. Abbasi, S. McManus, M. Payne, M. Barrick, J. Pulikottil, P. Bridges, and K. Schwan. Exploiting Latent I/O Asynchrony in Petascale Science Applications. *International Journal of High Performance Computing Applications*, 25(2):161–179, May 2010.
 - [14] Hongfeng Yu, J.H. Chen, Chaoli Wang, K. Ma, and R.W. Grout. A Study of In-Situ Visualization for Petascale Combustion Simulations. Technical report, 2009.